2-28-2002

Context Aggregation and Dissemination in Ubiquitous Computing Systems

Guanling Chen
Dartmouth College

David Kotz
Dartmouth College

Context Aggregation and Dissemination in Ubiquitous Computing Systems

Guanling Chen and David Kotz
Dartmouth College
{glchen, dfk}@cs.dartmouth.edu
http://www.cs.dartmouth.edu/~solar/

Dartmouth Computer Science Technical Report TR2002-420

February 28, 2002

Abstract

Many "ubiquitous computing" applications need a constant flow of information about their environment to be able to adapt to their changing context. To support these "context-aware" applications we propose a graph-based abstraction for collecting, aggregating, and disseminating context information. The abstraction models context information as *events*, produced by *sources* and flowing through a directed acyclic graph of event-processing *operators* and delivered to subscribing applications. Applications describe their desired event stream as a tree of operators that aggregate low-level context information published by existing sources into the high-level context information needed by the application. The operator graph is thus the dynamic combination of all applications' subscription trees.

In this paper, we motivate and describe our graph abstraction, and discuss a variety of critical design issues. We also sketch our Solar system, an implementation that represents one point in the design space for our graph abstraction.

1 Introduction

In a ubiquitous computing environment (sometimes called pervasive computing), in which a user may interact with dozens or hundreds of computationally enhanced devices, *user attention* becomes a scarce resource. It is unreasonable to expect a user to configure and manage these devices, particularly when the devices and their interactions change as the environment changes around them. "Ubicomp" applications must be aware of the context in which they run [29]. These *context-aware* applications can reduce user distraction by dynamically adjusting their behaviors to the current context, that is, the current state of the user, the current computational environment, and the current physical environment [30].

Context information is derived from an array of diverse information sources, such as location sensors, weather or traffic sensors, computer-network monitors, and the status of computational or human services. A fundamental challenge in ubiquitous computing, then, is to *collect* raw data from thousands of diverse sensors, *process* the data into context information, and *disseminate* the information to hundreds of diverse applications running on thousands of devices, while *scaling* to large numbers of sources, applications, and users, *securing* context information from unauthorized uses, and respecting individuals' *privacy*. In this paper we address this fundamental challenge by proposing a graph abstraction for context information collection, aggregation, and dissemination, and show how it meets the flexibility and scalability challenges. Its security and privacy features are beyond the scope of this paper.

We discuss the motivation and justification of the graph abstraction in Section 2. In Section 3 we describe the specifics of the graph abstraction. Section 4 discusses the many design decisions involved in realizing the graph abstraction, and Section 5 gives overview about the specific choices we made in our prototype "Solar system". We mention related work in Section 6 and summarize in Section 7.

2 Motivation

We arrived at our graph abstraction by considering the structure of applications that consume context informa-

This research has been supported by DARPA contract F30602-98-2-0107, by DoD MURI contract F49620-97-1-03821, by Microsoft Research, by the Cisco Systems University Research Program, and by the USENIX Scholars Program.

tion, the proper location for processing sensor data into context information, and structures that can encourage reuse of code and of derived context information. The structure must be flexible and extensible to meet the fundamental challenge of diversity. The structure must also be scalable. Figure 1 sketches an evolution of alternative structures.

A context-aware application attempts to adapt to its changing context by monitoring a variety of sensors. Figure 1a depicts an application receiving sensor data from three sources. The application runs on one platform, commonly a mobile or embedded host. The sensors are located in the infrastructure. This arrangement sends all of the sensor data across the network link to the application platform, and expects the application and its platform to be capable of transforming the raw data into the desired context information. In a situation with slow or unreliable networks, and low-capability mobile platforms, this arrangement is unworkable. With hundreds or thousands of applications and platforms sharing a network connection, it is impossible.

A common approach is to construct a "context service" that receives all of the raw source data, and supplies information about the current context, and changes to the context, to interested applications. (The "location service" seen in many systems is a special case of this approach.) Figure 1b shows that much of the processing has been moved off of the application platform, and may be shared by multiple applications. The context service provider defines the semantics of the context information it provides. While it is possible that the information meets the needs of some applications, in general the applications must process the output of the context service.

Alternatively, the application could push its application-specific processing into the network as a proxy, essentially an application-specific context service. Figure 1c demonstrates this approach. Note, however, that there will be one application-specific proxy for each application, which does not scale well.

We need a compromise that encourages sharing of fundamental transformations of sensor data into context information, but allows application-specific operations in the network. One possibility (not shown) is to supply a shared context service and install a proxy for each application. Figure 1d takes this approach one step further, decomposing the context service into smaller modules that produce context information of various types and forms. Application-specific proxies may now select the most appropriate inputs to begin their processing.

Finally, in Figure 1e we see that when there are many applications needing context information, they may be able to share both the application-specific as well as the generic processing steps. In the next section, we call this abstraction an *operator graph*. The burden of converting source data into context information is on servers in the network, not on application platforms. The decomposed graph structure improves flexibility, compared to a monolithic context service, and improves scalability, by avoiding a centralized context service, avoiding the transmission of unnecessary data to application platforms, and by sharing context processing across applications wherever possible.

3 The abstraction

Ubicomp researchers have long recognized the need for context collection, aggregation, and dissemination [9, 12, 32]. The challenge is to allow applications to define their own operations, to describe flexible compositions of operations, and to support many such applications with scalable performance. Based on our observations in the preceding section, we propose an abstraction for context collection, aggregation, and dissemination based on a directed acyclic graph (DAG). This abstraction can meet the fundamental challenges of flexibility, scalability, and security, although a discussion of security is beyond the scope of this paper.

In this section, we introduce the operator-graph abstraction. Then we classify several types of commonly used operators and sketch an example operator graph for an office scenario. Finally, we discuss the subtle semantics of operator state and "one-time" subscription requests.

3.1 Events, operators, and graphs

Context-aware applications respond to context changes by adapting to the new context. These applications are likely to have an "event-driven" structure, where context changes are represented as *events*. In our graph abstraction, then, we represent context information as events.

We treat sensors of contextual data as *information sources*, whether they sense physical properties such as location, or computational properties such as network bandwidth. Information sources produce their data as events. The sequence of events produced are an *event stream*, which is inherently unidirectional. An event *publisher* produces an event stream, and an event *subscriber* consumes an event stream.[1]

An *operator* is an object that subscribes to and processes one or more input event streams, and publishes another event stream. Since the inputs and output of an operator are all event streams, the operators can be connected recursively to form a directed acyclic graph, an event-flow graph that we call the *operator graph*.

[1] Notice that there is a one-to-one relationship between publishers and event streams. In some other event systems, more than one entity may publish events into an event stream.

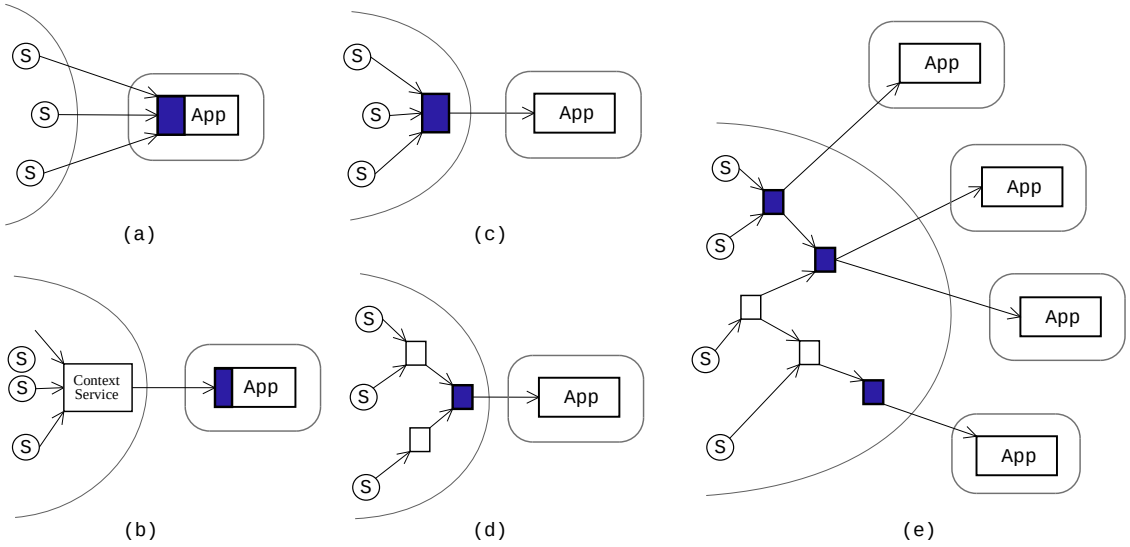

Figure 1: The circles are information sources, the white squares are operators, and the dark rectangles represent application-specific processing. (a) Send raw data from the sources to the application, which converts the data into the context information it needs. (b) A "context service," receives all raw source data, and provides higher-level context information to applications, but some application-specific processing is still necessary. (c) Push the application-specific processing into the network as a proxy. (d) Decompose the processing into application-independent portions and application-specific portions. (e) Allow multiple applications to share data streams where possible.

Our operator graph consists of three kinds of nodes: sources, operators, and applications. The *sources* have no subscriptions. They are wrappers for context sensors. *Operators* are deterministic functions of their input events. They only publish an event when they receive an input event. *Applications* are sinks of the graph. They subscribe to one or more event streams and react to incoming events (and possibly other stimuli, such as interactions with the user).

In our operator graph, a directed edge from node A to node B represents that node B subscribes to the event stream published by node A. The operator graph may not be a tree because an operator may subscribe to multiple streams, and its published output stream may have more than one subscriber. In summary, the *publishers* in the graph are the sources and operators, and the *subscribers* in the graph are the operators and applications.

There are four common categories of operators (see Figure 2). A *filter* outputs a subset of its input events. (For example, a sensor publishes the temperature every 10 seconds while one application needs alerts only when the reading exceeds 90 degrees.) A *transformer* inputs events of type E1 and outputs events of type E2. E2 may be the same as E1 if the transformer only changes some attribute values. (For example, a location sensor reports coordinates, but the application needs a symbolic value such as "Lobby.") The *merger* simply outputs every event it receives. (For example, an active-map application that displays the current location of all employees merges the readings from all location sensors.) While mergers are not strictly necessary, since any of the merger's subscribers could directly subscribe to the same inputs, a merger aids re-use of event streams. An *aggregator* outputs an arbitrary type event stream based on the events in one or more input event streams. (For example, a "max-min thermometer" operator outputs an event when it detects a new maximum or new minimum on its input stream of current temperature readings.)

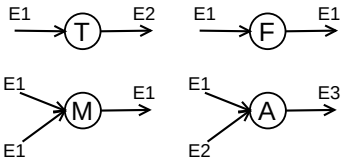

Figure 2: Four types of operators: T as Transformer, F as Filter, M as Merger, and A as Aggregator.

3.2 An example operator graph

Figure 3 presents an example operator graph to show how the raw events from information sources flow through the operators to become directly usable by the applications. Circles represent event publishers; the letter inside indicates its category (S stands for source). Squares represent applications that consume the events.

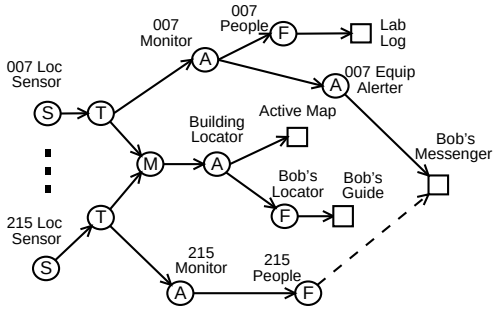

Figure 3: An example operator graph.

Suppose we have location-tracking sensors installed in each room and badges attached to people and devices. Each time a sensor detects a signal from a badge, it sends out an event containing the badge ID and the timestamp. In the figure these sources are labeled "Loc Sensor" with a room number; each has a transforming operator to map the badge ID to the person or device's name associated with it.

The *Building Locator* operator subscribes to the current location of every badge, based on the transformed and merged events that originate from the location sensors. It records the current location in its internal state. (We discuss stateful operators below.) It generates a "location change" event whenever it sees a badge change location. This output event stream can be used by the *Active Map* application (such as [27]) to display the badges' current location in real time. Another subscriber, *Bob's Locator*, filters for changes in Bob's location. Using this information, a *Guide* application [1, 11] running on Bob's PDA can display information related to his current location.

Another reasonable structure, not shown, is to first merge the events from all location sensors and then transform them using only one transformer, to which the Building Locator subscribes. Any application that cares about location events only in one particular room can filter the Building Locator's output. Although that approach seems awkward, it allows the Building Locator to resolve sensor conflicts (where multiple location sensors report seeing a badge at the same time).

Returning to our example, the operator *007 Monitor* tracks the set of badges currently in the lab. When a new badge is sensed, it generates a "badge entering" event. When a badge has not been sensed in the past few sensor reports, this operator outputs a "badge leaving" event. The filter *007 People* emits events about people only, not devices. The application *Lab Log* subscribes to that event stream and records the events with timestamp for future reference.

If the *007 Equipment Alerter* receives a "leaving" event for certain equipment, without receiving a "leaving" event for authorized personnel at about the same time, it publishes an alarm event that should be sent to the lab administrator (Bob), whose *Messenger* application displays these alarms on his PDA. If there is nobody in the room with Bob, the Messenger beeps and displays the message. If there are other people in the room, the Messenger vibrates instead. Notice the Messenger subscribes to "215 People" operator (the dashed arrow) because Bob is in room 215 now. This subscription is dynamic and will change as Bob moves around. We discuss the concept of context-sensitive subscriptions in Section 3.5.

There are several advantages of the operator graph abstraction. First, applications receive events semantically closer to their needs than those produced by the sources. Second, due to the modular, object-oriented design we benefit from operator reusability, data abstraction, and maintainability. Third, due to the modular design this operator graph can be deployed across a network and achieve the benefits of parallelism and distribution. Fourth, since filters and aggregators can dramatically reduce traffic along the graph edges, they reduce inter-process (and often inter-host) communication requirements. Finally, by sharing the common operators and event streams the system can support more such applications and more users.

3.3 Operator state

Many operators need to keep internal state information to be used when processing events. The state may be simple, as in an aggregator that simply records the previous event to detect changes. The state may be complex, as in an operator that tracks the current location of many users or the current value of every stock on the market. Filter, transformation, and merger operators are stateless; aggregators may have state.

Our graph abstraction allows the subscriber to choose one of two possible semantics for a new subscription to a stateful operator: 1) the subscription is treated as for stateless operators, or 2) the operator should "push" its current state to the subscriber before any new events are published. In the latter semantics the operator publishes a special sequence of events to the new subscriber only, events that are marked as "state-pushing events" and when considered together represent the current state of the operator. (This feature is reminiscent of the Gryphon expansion operation [3].)

Consider Figure 3. The 007 Monitor maintains a list of badges currently in the lab and publishes changes to this list. The Lab Log logs all the change events, and never needs the original state. The Active Map, on the other hand, needs a "state push" when it first subscribes to the Building Locator, so it can properly locate slow-moving devices like printers.

3.4 One-time subscription requests

The operator graph is an event-oriented abstraction that has publish-and-subscribe interfaces for disseminating information to applications. Occasionally an application may not need the ongoing event stream, but simply needs to obtain the current value. In another system, the application might query the information source. In the operator graph we retain the publish-and-subscribe abstraction by permitting "one-time" subscriptions of stateful operators. An application that needs to obtain the current value of the information published by an operator makes a one-time subscription to that operator. The operator "pushes" its state, as described above, and then cancels the subscription.

The one-time subscription approach has several advantages, largely resulting from its simplicity. There is only one abstraction: publish and subscribe, which streams events from publisher to subscriber. This simplicity avoids the need for additional interfaces and maintains the unidirectional data flow. The subscriber's control flow remains event-oriented rather than blocking for the results of a query. The programmer of the subscriber can choose one-time or permanent subscriptions based on their needs. The programmer of the publisher need not know anything about queries or one-time subscriptions, only about state push.

3.5 Context-sensitive subscriptions

Many context-aware applications use one aspect of the context (such as the user's location) to subscribe to other information about that context (such as the set of people, devices, or sensors in that location). As the user changes location, the application must cancel its subscriptions, then locate and subscribe to appropriate sources for the new location. These location-sensitive subscriptions are a general case of what we call *context-sensitive subscriptions*.

It is possible for the application to actively monitor the user's location (for example), and when the user moves to manually adjust its other subscriptions. To reduce programmer effort and to avoid redundant monitoring of the same context by many applications, we aim to support context-sensitive subscriptions directly in the infrastructure. From the viewpoint of the graph, the links representing a context-sensitive subscription are dynamic and the events may flow through different paths as the context changes. We discuss the potential for context-sensitive names to represent context-sensitive subscriptions in Sections 4.2 and 5.

4 Design space

There are many design issues involved in realizing the operator-space abstraction. In this section we consider the representation of events, how to name operators or event streams, how to route events from publishers to subscribers, and the operator programming model.

4.1 Event representation

Events are passed from publisher to subscriber, typically across a network connection of some kind. Ultimately, any representation agreeable to both publisher and subscriber will work, but there are three typical approaches used by event-distribution systems. The event may be a typed object appropriate to a particular object-oriented language, a set of attribute-value pairs (usually represented as lines in an ASCII string), or more recently a small XML document. Each representation has advantages and disadvantages.

An object representation allows the event to include a complex data structure, if desired. The inherent type hierarchy an object-oriented language can be used for type-checking when matching subscribers to publishers. Furthermore, type inheritance allows subscribers to process a general class of events even when publishers may send events of more refined subclasses. For example, a location aggregator can receive any location event regardless of whether it is a GPSLocationEvent, ActiveBadgeLocationEvent, or CricketLocationEvent, if they are all subclasses of LocationEvent. On the other hand, an object representation is usually tied to a particular language, such as Java.

An attribute-value representation is typically more limited than an object representation, although some hierarchical representations (such as INS [2]) do provide structure, and the use of wildcards and implicit fields provide a limited form of inheritance. The simpler representation greatly facilitates content-based event routing (see Section 4.3), and is language- and platform-independent.

Although XML encoding provides more opportunities for structure, XML adds substantial overhead. Parsing every incoming event, constructing every outgoing event, and transmitting information in the verbose XML format, reduces event throughput and adds substantial latency along the event flow. While compressing the XML reduces bandwidth consumption [33], it adds more processing overhead.

4.2 Operator naming

A primary feature of the graph abstraction is the opportunity to re-use the event streams between applications and between users. It is always possible for an application to construct its event flow from basic sources and a tree

of generic and custom operators. When an application describes such a tree and asks the infrastructure to deploy the operators, the supporting infrastructure can match the new description against the existing graph to identify whether any existing event streams can be used to satisfy all or part of the new request. To make life easier for application programmers, however, it would be helpful if common event streams could be constructed and named by an administrator, or by other users, and then new applications can subscribe to these event streams by name.

So, we need a method to name event streams (or equivalently, the operator that publishes the stream). Incidentally, we must also name sources. There are many possible approaches to naming.

The name space could be organized as a tree, as in many file systems. For those publishers given names, the name describes a path from the root to a leaf in the tree. For example, a temperature sensor in Sudikoff room 215 might be named [/Sudikoff/2F/215/temp-sensor/]. To enhance scalability, multiple levels of naming may be helpful; although many examples exist [25, 4], perhaps the most common example is the two-level name (hostname:filename) used in URLs.

The name space may be less structured. Each named publisher could be given a set of descriptive attribute-value pairs [2, 18]. The above temperature sensor might be named [sensor=temperature, room=215, floor=2, building=Sudikoff].

It is arguable whether one approach has clear advantages over the other. In either case the name should be a descriptive handle. In one case the description is a tuple of attributes and values, and in the other case the same attributes may be implicit in the structure of the tree. Both depend heavily on conventions that define the names of the attributes (or structure of the tree) and the range of values (or names of tree links). The conventions used to structure the tree are likely stricter than those in a set of attributes, which may make the tree less attractive in a dynamic ubicomp environment.

Another important role for naming is to facilitate resource discovery. In tree-based names a wildcard allows an application to easily describe a large set of publishers, e.g., [/Sudikoff/*/*/temp-sensor/]. The same effect might be obtained in an attribute-based system that allows partial matches, e.g., [sensor=temperature, building=Sudikoff].

We are intrigued by the potential for *context-sensitive names*, that is, names whose binding changes when the context changes. For example, [/people/profs/Bob/location/temp-sensor] might refer to the temperature sensor in the same room as Bob. A subscription to that name would dynamically be rebound to a subscription to the appropriate publisher when Bob changes location. It is not clear how to encode this level of indirection in an attribute-based approach.

Finally, it is a challenge to implement a large name space efficiently. The tree structure leads to efficient name resolution but may encounter a bottleneck at the root. Some structural conventions may be imposed on an attribute-based approach to improve resolution scalability [2]. While some recent peer-to-peer systems hash a *full* name as the first step in locating the object associated with a name [10, 13, 23], it is not clear how that approach might support wildcards and context-sensitive names.

4.3 Event routing

Although the graph abstraction links publishers directly to subscribers, routing events from a publisher to its set of subscribers is essentially a multicast problem that may be implemented in many ways. The simplest approach is to use unicast, to send a copy of the event to each subscriber. This approach will not scale when there are many subscribers. Where IP multicast is supported, applications might subscribe and unsubscribe to event streams by joining or leaving particular multicast groups. This approach requires one IP-multicast group for each publisher, however, which is not scalable. An overlay network can use an application-level multicast protocol among a set of servers acting as multicast routers, based either on traditional multicast groups [20] or based on groups defined by names [2].

We can take the multicast concept one step farther to content-based event routing [3, 7, 33]. These systems also use an overlay network of servers, often called *brokers*, which route events to subscribers based on the content (attributes) of the events, not simply based on the destination group or name. In effect, all publishers send events to the global event stream, and subscribers describe the events they want to receive as filters. Siena filters can even recognize event sequences. Some of the systems can encode simple transform operations.

Since event brokers implement simple merge, filter, and transform operators, it is tempting to add complex operators like aggregators. Since the brokers are essentially pattern-matching engines, it is unclear whether they might be extended to implement complex operators. All such systems must balance expressiveness with scalability [6].

A reasonable compromise is to use content-based routing as the routing substrate of the operator graph. Implement merge and simple filter operations in the event broker layer, and the more complicated operators remain independent operators that subscribe to the routing system and publish events back into the routing system.

4.4 Programming model

What abstraction do we provide to the application programmer? The programmer may deal explicitly with the

operator-graph abstraction, composing an event stream by describing an operator tree based in named sources, and using generic and custom operator classes. Or, the programmer may describe the desired events from named sources using a descriptive higher-level language, which is translated by a compiler into the appropriate operator tree.

The *explicit* approach exposes the operator-tree abstraction to the programmer. The programmer manually derives an event flow that produces context information from named event streams using generic and custom operator classes. Then she uses a subscription language to describe the structure of the tree. She may optionally name the intermediate or final event stream, for others to use.

We speculate that the act of manually deriving an event flow, and the temptation to use existing operator classes where available, will encourage programmers to derive similar trees in similar situations, increasing the opportunities for re-use of event streams. Similarly, programmers will be likely to name event streams for use by other users or in other applications.

With a more complex subscription language, the operator-tree abstraction may be *transparent* to the programmer. Using the language to describe the aggregation of events from named publishers into the desired context, the programmer encodes all of the necessary logic in one program. A compiler translates the subscription into a tree of operators, which is deployed in the same way as in the explicit programming model. The challenge is to invent a subscription language sufficiently powerful to encode complicated aggregations, and yet simple enough to efficiently parse into an operator tree. A language like Java is highly expressive, but a language like SQL or XQuery may offer more structure. The language needs a structure that encourages the programmer to describe the event flow in a way that is easily decomposable and likely to match other applications' operator trees. With sophisticated compiler analysis, it may be possible to define finer-grain operators and support finer-grain sharing within the graph. With knowledge of the semantics, the compiler may also be possible to rearrange operator trees to allow matches that would otherwise not have occurred.

4.5 Summary

We discuss several design issues for realizing the operator-graph abstraction. The combination of different choices will result in different systems. For example, if events are represented as XML documents, the application programmer might use the XQuery programming language to describe its subscription for a compiler to translate into an operator tree and a set of filter expressions to provide to an XML-based event-routing overlay network.

In next section we discuss the prototype of our Solar system, which represents another combination of design choices. While Solar has many interesting characteristics, we hope to explore some of the other design choices in related prototypes.

5 The Solar system

We are building a prototype infrastructure for context collection, aggregation, and dissemination, based on the operator-graph abstraction. We describe our prototype, the "Solar" system, in detail in a technical report [8]. In this paper we focus on the graph abstraction and related design choices.

Our Solar prototype is implemented in Java. It models events as Java objects and uses Java serialization for event transmission. The operators are small Java objects that implement a simple publish/subscribe interface.

Solar names sources and operators in a tree-structured name space. Thus, operators have path names like [/Sudikoff/2F/215/temp-sensor]. We extend the tree abstraction with two types of *soft links*. *Alias* nodes bind one name to another. Unlike a Unix symbolic link, however, most alias nodes are designed to change their binding when certain context changes. Thus, the alias [/people/profs/Bob/location] may be bound to [/Sudikoff/2F/215] now, then later to [/Sudikoff/0F/007] when Bob walks to room 007. *Dynamic directories* dynamically compute their set of children based on context. For example, [/Sudikoff/0F/007/people] is a directory whose children are other nodes in the tree, nodes that represent people. The combination can be quite powerful; for example, [/people/profs/Bob/location/people], is a directory containing a list of people co-located with Bob. The list changes when Bob moves or when people enter or leave Bob's current room.

Solar's naming conventions thus encode context in the name space. Dynamic context is captured with the dynamic soft links. To enable applications to monitor changes to the namespace, and hence changes to the encoded context, all namespace nodes are publishers. Directories publish changes to their set of children, and aliases publish changes to their binding. Aliases and dynamic directories are operators that subscribe to the context information necessary to change their binding or set of children.

Given this context-sensitive namespace, we then encourage applications to subscribe to context-sensitive names. Thus, an application desiring to track the set of people Bob meets in a given day, subscribes to the operator at name [/people/profs/Bob/location/people], and receives an event about changes to the set of people surrounding Bob. We show below how Solar supports subscriptions to context-sensitive names.

At the center of any Solar system is a *Star*, which keeps a reference to the root of naming tree, maintains the operator graph, and services requests for new subscriptions. When the Star receives a new subscription-tree description, it parses the description, checks the name space, and matches the subscription tree against its internal data structure representing the operator graph. When it decides to deploy an operator, it instantiates the operator's object on one of many *Planets*. Each Planet is an execution platform for Solar sources and operators. Applications run outside the Solar system and use a small Solar library that allows them to send requests to the Star, and to manage their subscriptions, over standard network protocols.

Planets play a key role in the subscriptions of resident operators. When deploying new subscriptions, the Star tells the Planets to record a subscription from one of its operators to an operator in another Planet. In our implementation there is at most one network (TCP/IP) connection between any two Planets, regardless of the number of operators on or subscriptions between the two Planets.

Planets support subscription requests that involve context-sensitive names (CSNs). These subscription requests are mapped to subscriptions, which need to be changed when the CSN binding changes. Consider an operator X that records the name of every person Bob meets. The operator requests subscription to the CSN [/people/profs/Bob/location/people], currently bound to an operator P. X's Planet subscribes to the name [/people/profs/Bob/location]. The Planet receives the current binding and subscribes X to P. When Bob moves, suppose the binding changes to operator Q. X's Planet contacts P's Planet to remove X from P's data structure, and contacts Q's Planet to add X to the Q's data structure. All the work is done by planets and the namespace operators; P, Q, and X are never involved.

These and other aspects of the Solar architecture are key to its scalability and flexibility. Our next step is to experiment with the use of Solar in several real-world context-sensitive mobile applications to determine the value of the abstraction and the performance of the system. We installed an IR-based location system from Versus Technologies,[2] to supply location context to our Solar system and its applications. We plan to add more information sources to enrich the context space and to explore the performance and flexibility of the operator graph abstraction. Finally, Solar has unique mechanisms for access control and authorization, but their description is beyond the scope of this paper.

[2] http://www.versustech.com/

6 Related work

Many have studied context-aware applications and their support systems. In Xerox Parc's distributed architecture each user's "agent" collects context (location) about that user, and decides to whom the context can be delivered based on that user's policy [31, 34]. AT&T Laboratories at Cambridge built a dense network of location sensors to maintain a world model shared between users and applications [17]. Location context can be used to select on-the-spot information for tourist guide applications [1, 11]. HP's Cooltown project adds Web context to the environment by allowing mobile users to receive URLs sent by ubiquitous beacons [22]. HP's Easyliving focuses on a smart space that is aware of the user's presence and adjusts environmental settings to suit her needs [5].

A few projects specifically address the flexibility and scalability of context aggregation and dissemination. Like Solar, the Context Toolkit is also a distributed architecture supporting context fusion and delivery [12]. It uses a *widget* to wrap a sensor, through which the sensor can be queried about its state or activated. Applications can subscribe to pre-defined aggregators that compute commonly used context. Solar allows applications to dynamically insert operators into the system and compose refined context that can be shared by other applications. The Context Toolkit allows applications to supply filters for their subscriptions, while Solar introduces general filter operators to maintain a simple abstraction.

Given the type of desired data, some systems automatically construct a data-flow path from sources to requesting applications, by selecting and chaining appropriate components from a system repository [21, 19]. CANS can further replace or rearrange the components to adapt to changes in resource usage [16]. To apply this approach to support context-aware applications, the system manager must foresee the necessary event transformations and install them in the component repository. These systems offer no specific support for applications to provide custom operators. Active Names, on the other hand, allow clients to supply a chain of generic or custom components through which the data from a service must pass [35]. Also, Active Streams support event-oriented interprocess communication, and allow application-supplied *streamlets* to be dynamically inserted into the data path [14].

All of these approaches encourage the re-use of standard components to construct custom event flows. None, to our knowledge, specifically encourage the dynamic and transparent re-use of event streams across applications and users. Solar's re-use of operator instances, and their event streams, avoids redundant computation and data transmission, and improves scalability.

Solar is designed to support a wide variety of sensor

data, including computational as well as physical parameters. Solar may then be the delivery mechanism for systems that allow mobile applications to adapt to changes in computational resources. For example, Odyssey applications are aware of the state of resources and can adapt to variations in bandwidth [28] and battery power [15].

As we discuss in Section 4.3, there are many options for event routing. Solar currently uses a point-to-point links between a publisher and its subscribers. Although the implementation multiplexes links on Planet-to-Planet socket connections, and implements multicast within a Planet, we may eventually construct an overlay multicast network on the Planets. We may also consider using content-based event routing [7, 3, 33] to support the operator graph. Ultimately, we need to evaluate whether Solar's explicit filter operators will be more or less efficient than the implicit filtering in content-based event routing system like Siena.

Solar names many of its publishers. We may be able to use names instead of addresses for routing [24, 2]. Peer-to-peer systems typically hash the full text name into an ID for the purpose of routing [10, 13, 23]. These approaches do not easily support context-sensitive subscriptions, however.

7 Summary

To support context-aware mobile applications, we propose a graph-based abstraction for context aggregation and dissemination. The abstraction models the contextual information sources as event publishers. The events flow through a graph of event-processing operators and become customized context for individual applications. This graph-based structure is motivated by the observation that context-aware applications have diverse needs, requiring application-specific production of context information from source data. On the other hand, applications do not have *unique* needs, so we expect there is substantial opportunity to share some of the processing between applications or users. The situation calls for both flexibility and scalability, and our proposed operator-graph abstraction meets both challenges. It allows the flexible construction of event streams through composition of generic and custom operators. It encourages scalability through re-use of event streams across applications and users wherever possible, by migrating the load off weak mobile application platforms and into powerful network servers, and by distributing that load among many network servers.

We discuss the details of the operator graph abstraction, the four types of the operators, the semantics of stateful operators and one-time subscriptions, and the context-sensitive subscriptions that make the graph dynamic. There are many ways to realize the graph abstraction in a supporting system, and we discuss design issues involved with event representation, operator naming, event routing, and programming models. We give an overview of our Solar system that implements the graph abstraction. Interested readers can find more details in technical reports about Solar [8] and about a smart reminder application built on top of Solar [26].

References

[1] G. D. Abowd, C. G. Atkeson, J. Hong, S. Long, R. Kooper, and M. Pinkerton. Cyberguide: A mobile context-aware tour guide. *Wireless Networks*, 3(5):421–433, October 1997.

[2] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 186–201, Kiawah Island Resort, South Carolina, December 1999.

[3] G. Banavar, M. Kaplan, K. Shaw, R. E. Strom, D. C. Sturman, and W. Tao. Information flow based event distribution middleware. In *Proceedings of the Middleware Workshop at the 19th IEEE International Conference on Distributed Computing Systems*, Austin, Texas, May 1999. IEEE Computer Society Press.

[4] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine: An exercise in distributed computing. *Communication of ACM*, 25(4):260–274, April 1982.

[5] B. Brumitt, B. Meyers, J. Krumm, A. Kern, and S. Shafer. EasyLiving: Technologies for intelligent environments. In *Proceedings of the Second International Symposium on Handheld and Ubiquitous Computing*, pages 12–29, Bristol, UK, September 2000. Springer-Verlag.

[6] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an Internet-scale event notification service. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 219–227, Portland OR, USA, July 2000.

[7] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[8] G. Chen and D. Kotz. Supporting adaptive ubiquitous applications with the SOLAR system. Technical Report TR2001-397, Dept. of Computer Science, Dartmouth College, May 2001.

[9] N. H. Cohen, A. Purakayastha, J. Turek, L. Wong, and D. Yeh. Challenges in flexible aggregation of pervasive data. Technical Report RC21942, IBM TJ Watson Research Center, January 2001.

[10] F. Dabek, E. Brunskill, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica, and H. Balakrishnan. Building peer-to-peer systems with Chord, a distributed lookup service. In *Proceedings of the 8th Annual Workshop on Hot Topics in Operating Systems*, Elmau, Germany, May 2001.

[11] N. Davies, K. Cheverst, K. Mitchell, and A. Friday. Caches in the air: Disseminating tourist information in the GUIDE

system. In *Proceedings of the Second IEEE Workshop on Mobile Computing Systems and Applications*, New Orleans, Louisiana, February 1999.

[12] A. K. Dey. *Providing Architectural Support for Building Context-Aware Applications*. PhD thesis, College of Computing, Georgia Institute of Technology, December 2000.

[13] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 8th Annual Workshop on Hot Topics in Operating Systems*, Elmau, Germany, May 2001.

[14] G. Eisenhauer, F. E. Bustamante, and K. Schwan. A middleware toolkit for client-initiated service specialization. *Operating Systems Review*, 35(2):7–20, April 2001.

[15] J. Flinn and M. Satyanarayanan. PowerScope: A tool for profiling the energy usage of mobile applications. In *Proceedings of the Second IEEE Workshop on Mobile Computing Systems and Applications*, New Orleans, Louisiana, February 1999. IEEE Computer Society Press.

[16] X. Fu, W. Shi, A. Akkerman, and V. Karamcheti. CANS: Composable, adaptive network services infrastructure. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, San Francisco, California, March 2001. USENIX.

[17] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster. The anatomy of a context-aware application. In *Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 59–68, Seattle, WA, August 1999.

[18] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 146–159, Chateau Lake Louise, Canada, October 2001.

[19] J. I. Hong and J. A. Landay. An infrastructure approach to context-aware computing. *Human-Computer Interaction*, 16(2&3), 2001.

[20] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole, Jr. Overcast: Reliable multicasting with an overlay network. In *Proceedings of Forth Symposium on Operating Systems Design and Implementation*, pages 197–212, San Diego, CA, October 2000.

[21] E. Kiciman and A. Fox. Using dynamic mediation to integrate COTS entities in a ubiquitous computing environment. In *Proceedings of the Second International Symposium on Handheld and Ubiquitous Computing*, pages 211–226, Bristol, UK, September 2000. Springer-Verlag.

[22] T. Kindberg, J. Barton, J. Morgan, G. Becker, D. Caswell, P. Debaty, G. Gopal, M. Frid, V. Krishnan, H. Morris, J. Schettino, and B. Se. People, places, things: Web presence for the real world. In *Proceedings of the Thrid IEEE Workshop on Mobile Computing Systems and Applications*, pages 19–28, Monterey, California, December 2000. IEEE Computer Society Press.

[23] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–201, Cambridge, MA, November 2000.

[24] H. T. Kung. MotusNet: A content network. 2001.

[25] B. W. Lampson. Designing a global name service. In *Proceedings of the 4th ACM Symposium on Principles of Distributed Computing*, pages 1–10, Minaki, Ontario, 1986.

[26] A. Mathias. SmartReminder: A case study on context-sensitive applications. Technical Report TR2001-392, Dept. of Computer Science, Dartmouth College, June 2001. Senior Honors Thesis.

[27] J. F. McCarthy and E. S. Meidel. ACTIVE MAP: A visualization tool for location awareness to support informal interactions. In *Proceedings of First International Symposium on Handheld and Ubiquitous Computing*, pages 158–170, Karlsruhe, Germany, September 1999.

[28] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 276–287, Saint-Malo, France, October 1997.

[29] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8(4):10–17, August 2001.

[30] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *Proceedings of IEEE Workshop on Mobile Computing Systems and Applications*, pages 85–90, Santa Cruz, California, December 1994. IEEE Computer Society Press.

[31] W. N. Schilit. *A System Architecture for Context-Aware Mobile Computing*. PhD thesis, Columbia University, May 1995.

[32] A. Schmidt, K. A. Aidoo, A. Takaluoma, U. Tuomela, K. V. Laerhoven, and W. V. de Velde. Advanced interaction in context. In *Proceedings of First International Symposium on Handheld and Ubiquitous Computing*, pages 89–101, Karlsruhe, Germany, September 1999.

[33] A. C. Snoeren, K. Conley, and D. K. Gifford. Mesh based content routing using XML. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 160–173, Chateau Lake Louise, Canada, October 2001.

[34] M. Spreitzer and M. Theimer. Providing location information in a ubiquitous computing environment. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 270–283, December 1993.

[35] A. Vahdat, M. Dahlin, T. Anderson, and A. Aggarwal. Active Names: Flexible location and transport of wide-area resources. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, Colorado, October 1999. USENIX.